THE UNIVERSITY OF MELBOURNE

COMP 90048 Declarative Programming Workshop 11 (week12)

2019 semester 1

by Wendy Zeng

Tutorial : Tue 18:15 - 19:15 221 Bouverie St, room B113

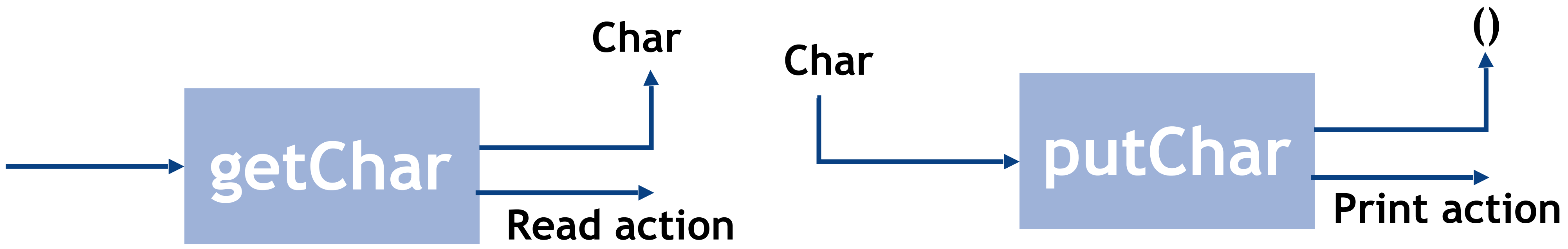
Wed 17:15 - 18:15 201 Bouverie St, room B132

Outline

1. More on Monad
2. Lazy Evaluation
   a. Case Study 1: Fibonacci Sequence
   b. Case Study 2: Merge Sort in Haskell
3. Quick Tips on WorkshopExtra
4. Subject Wrap Up

1. More on Monad

- IO:



- Bind operator in IO:

(>>=)   action1 >>= \x -> action2

1. *Perform* IO action 1, which produces (unwrap from the IO box) value x (Char/String/() if there's no value produced)
2. Create a new IO type using value of x, by \x -> action2
3. *Perform* IO action 2, and return the output of action 2

Evaluate an IO action has no side effect, but performing an IO action does

1. More on Monad

- Example1: *print_mtree :: Show a => Mtree a -> IO ()* (print out Mtrees with indentation showing the structure)

  data Mtree a = Mnode a [Mtree a]

  Solution 1: Traverse the Mtree structure, convert to string and perform IO at the same time

Mtree a → IO ()

```
to_io :: Show a => Int -> Mtree a -> IO()
to_io i (Mnode val children) =
    let val_to_io = putStrLn $ (replicate i ' ') ++ (show val)
          children_to_io = map (to_io (i+1)) children
    in
       val_to_io >>
       foldl (>>) (return ()) children_to_io
```

val_to_io and children_to_io: evaluate and define the IO action

Perform these IO in order by sequencing them (>>)

```
print_mtree1 :: Show a => Mtree a -> IO ()
print_mtree1 = to_io 0
```

1. More on Monad

- **Example1:** *print_mtree :: Show a => Mtree a -> IO ()* (print out Mtrees with indentation showing the structure)

  Solution 2: Traverse the Mtree structure and store a list of strings, then perform IO upon each string in order

Mtree a → [String] → IO ()

```
to_string :: Show a => Mtree a -> [String]
to_string (Mnode val children) = show val : children_to_strings
   where children_to_strings = map (' ':) (concatMap to_string children)

print_mtree2 :: Show a => Mtree a -> IO ()
print_mtree2 t =
   foldl (\acc str -> acc >> putStrLn str) (return()) (to_string t)
```

Perform these IO in order by sequencing them (>>)

1. More on Monad

- Example1: *print_mtree :: Show a => Mtree a -> IO ()* (print out Mtrees with indentation showing the structure)

  Solution 3: Traverse the Mtree structure and store a list of IO action, then perform each IO action in order

Mtree a → [IO ()] → IO ()

```
to_io_list :: Show a => Mtree a -> [IO ()]
to_io_list (Mnode val children) = print val : children_to_ios
   where children_to_ios = map (putChar ' ' >>) (concatMap to_io_list children)

print_mtree3 :: Show a => Mtree a -> IO ()
print_mtree3 t = foldl (>>) (return ()) (to_io_list t)
```

Or use fold1:

```
foldl1 (>>) (to_io_list t)
```

2. Lazy Evaluation

- Strictness:
  - Is a type of evaluation strategy, determined by the compiler
  - Non-strict: the expression may or may not be evaluated (GHC)
  - Strict: the expression is definitely evaluated (C, Java compiler etc.)
- Lazy Evaluation:
  - Is a way to implement non-strictness
  - func arg
    - In strict evaluation: does the arg evaluation first
    - In lazy evaluation: does the func application first, delaying the evaluation of arg

square (1+2)

Strict evaluation:

1 + 2 = 3
3 * 3

VS

Lazy evaluation:

(1+2) * (1+2)

2. Lazy Evaluation

- Example2: *fibs :: Int -> [Integer]* (computes the first N number from Fibonacci Sequence)

```
fibs1' :: Integer -> Integer -> Int ->
[Integer]
fibs1' _ _ 0 = []
fibs1' f1 f2 n = (f1+f2) : fibs1' f2 (f1+f2)
(n-1)

fibs1 :: Int -> [Integer]
fibs1 0 = []
fibs1 1 = [0]
fibs1 n | n > 1 = 0:1:fibs1' 0 1 (n-2)
```

```
allfibs :: [Integer]
allfibs = 0 : 1 : zipWith (+) allfibs (tail
allfibs)

fibs2 :: Int -> [Integer]
fibs2 n = take n allfibs
```

allfibs will only be called once
Construct where it needs to be, lazy evaluation will resume from where it stops to keep computing other numbers needed

Both have time complexity O(n), but the right-hand-side version has a higher constant factor due to lazy evaluation

2. Lazy Evaluation

- Example3: *merge_sort :: [a] -> [a]* (bottom-up implementation of merge sort)

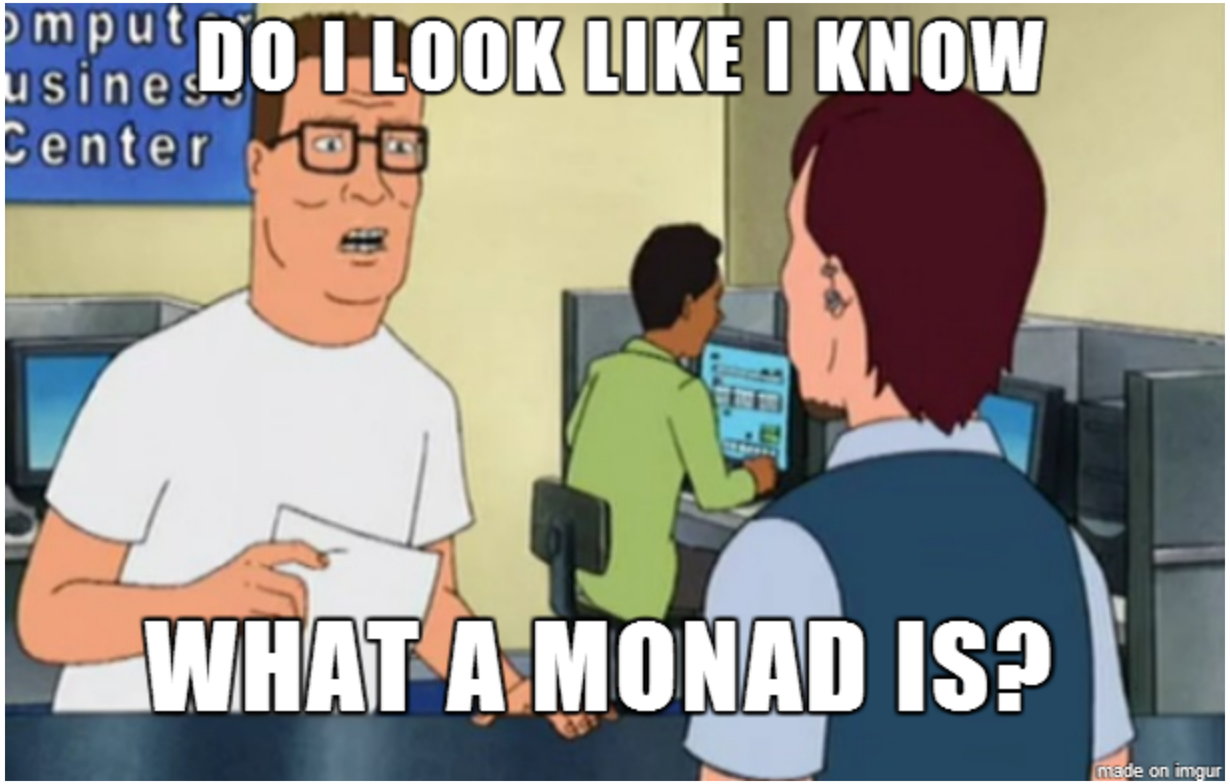
```
mergesort xs = repeat_merge_all (merge_consec (to_single_els
xs))

to_single_els [] = []
to_single_els (x:xs) = [x] : to_single_els xs

merge [] ys = ys
merge (x:xs) [] = x:xs
merge (x:xs) (y:ys)
      | x <= y = x : merge xs (y:ys)
      | x >  y = y : merge (x:xs) ys

merge_consec [] = []
merge_consec [xs] = [xs]
merge_consec (xs1:xs2:xss) = (merge xs1 xs2) : merge_consec
xss

repeat_merge_all [] = []
repeat_merge_all [xs] = xs
```

With lazy evaluation, single_to_els interleaves with merge_consec, once the two singleton lists are merged their space will be reclaimed which is beneficial for computation space

3. Quick Tips on WorkshopExgtra

- Attempt Q4, 5, 6, 7, 10, 11, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22
- Ignore everything to do with Mercury
- This gives you a fair bit of exercise in higher order functions beyond list and and their application on more complicated data structures
- List: Q 4-7, 19, 20
- BST: Q 10-11, Q 18
- Cord: Q 13, Q 17
  - data Cord a = Nil | Leaf a | Branch (Cord a) (Cord a)
- Double Linked List: Q 14
- Mtree, Ltree: Q 15-16
  - data Mtree a = Mnode a [Mtree a]
  - data Ltree a = LTLeaf a | LTBranch (Ltree a) (Ltree a)
- Queue: Q 21, 22

4. Subject Wrap Up



4. Subject Wrap Up

- Haskell
  - 1. Recursion:
    - Exhaustive and Inclusive
    - Pattern Matching
  - 2. Type Constructor and Data Constructor (ws2)
  - 3. Type Class
    - Ed, Ord, Read, Show, Num, Floating, Fractional
- Some examples to focus on:
  - BST exercises
  - Eval for pattern matching (WS 3)
  - Lecture slides examples

4. Subject Wrap Up

- Haskell
  - 4. Higher Order Functions:
    - map :: (a -> b) -> [a] -> [b]
    - filter :: (a -> Bool) -> [a] -> [a]
    - foldl :: (b -> a -> b) -> b -> [a] -> b
    - foldr :: (a -> b -> b) -> b -> [a] -> b
    - zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
    - concatMap :: (a -> [b]) -> [a] -> [b]
    - (.) :: (a -> b) -> (b -> c) -> a -> c
    - flip :: (a -> b -> c) -> b -> a -> c
    - any :: (a -> Bool) -> [a] -> Bool
    - all :: (a -> Bool) -> [a] -> Bool
  - 5. Partial Application and Currying (WS 5)
    - how you can write type signature for partially applied functions
  - 6. Monad
    - Maybe and IO (Do block in IO)
    - Return and >>= (>>)

4. Subject Wrap Up

- Prolog:
  - Prolog basics: semantics, rules & facts & queries, terms,
  - How unification works
  - Negation as failure
  - Examples to focus on:
    - List exercises
    - BST exercises
  - Tail recursion and how to use accumulator
  - All solutions (bagof and setoff)

THE UNIVERSITY OF MELBOURNE

Thank you

wendy.zeng@unimelb.edu.au

By Wendy Zeng